Concurrent programming notations in the object-oriented language Arche

Marc Benveniste, Valérie Issarny

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992
25ème anniversaire

N° 1822

Programme 1
Architectures parallèles, Bases de données,
réseaux et Systèmes distribués

CONCURRENT PROGRAMMING NOTATIONS IN THE OBJECT-ORIENTED LANGUAGE ARCHE

Marc BENVENISTE
Valérie ISSARNY

Décembre 1992

RR_1822

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Concurrent Programming Notations in the Object-Oriented Language Arche

Marc Benveniste*

Valérie Issarny

IRISA / INRIA-RENNES

Campus de Beaulieu, 35042 Rennes Cédex, FRANCE

{*mbenveni,issarny*}*@irisa.fr*

Publication Interne n°690- Décembre 1992, 34 pages

Programme 1

Abstract

Paradigms of object-oriented programming are attractive for the design of large distributed software. They notably provide a sound basis to develop applications that are easy to maintain and reuse. However, expressing concurrency in object-oriented languages raises some difficulties. For instance, integrating concurrency together with inheritance may lead to violate the encapsulation property of object-oriented programming.

In this paper, we present a new strongly-typed, concurrent object-oriented language, called Arche, that enables the development of reusable distributed software. The Arche language has been designed so as to keep all the benefits of the object paradigm. In particular, the Arche conditional synchronization mechanism is defined in such a way that synchronization constraints of an existing class can be specialized and reused. Furthermore, a mechanism that allows a larger scale synchronization, that is, involving more than just a pair of processes, is offered by supporting application of operations on a collection of objects. To our knowledge, such a facility has never been addressed in the framework of strongly-typed, concurrent object-oriented programming despite the ability to simply manage collection of peer objects it provides.

Key words: Concurrent object-oriented programming, inheritance, subtyping, conditional synchronization, type-states, synchronization-states, multi-party synchronization, multiprocedures, multi-operations.

---

*Supported by a grant from the XII CONACYT-CEFI fellowship program jointly held by MEXICO and FRANCE.

Notations de programmation concurrente dans le langage à objets Arche

Résumé

Les paradigmes de la programmation à objets sont attrayants pour la conception d'applications distribuées de grande taille. En particulier, ils fournissent une base solide pour développer des applications faciles à maintenir et à réutiliser. Néanmoins, l'expression de la concurrence dans les langages à objets n'est pas sans soulever des difficultés. Par exemple, intégrer à la fois les facilités de concurrence et d'héritage peut conduire à la transgression de la propriété d'encapsulation de la programmation à objets.

Dans cet article, nous présentons un nouveau langage concurrent à objets, fortement typé, appelé Arche, qui autorise le développement d'applications distribuées réutilisables. Le langage Arche a été conçu de manière à maintenir tous les avantages du paradigme objet. En particulier, le mécanisme de synchronisation conditionnelle de Arche est définie de telle sorte que les contraintes de synchronisation d'une classe existante peuvent être spécialisées et réutilisées. Par ailleurs, un mécanisme qui permet de synchroniser plus de deux processus est offert *via* l'application d'opérations sur une collection d'objets. A notre connaissance, une telle facilité n'a jamais été étudiée dans le cadre de la programmation concurrente à objets fortement typée malgré qu'elle permette de simplement gérer des collections d'objets similaires.

Mots clés : programmation concurrente à objets, héritage, sous-typage, synchronisation conditionnelle, états de type, états de synchronisation, synchronisation multi-partie, multiprocédures, multi-opérations.

1 Introduction

Besides mechanisms to deal with distribution, large distributed software design requires mechanisms for structuring data and algorithms. Furthermore, distributed applications should be easy to maintain and should exhibit qualitative features such as reliability and readability. Object-oriented programming fulfills many of the above *desiderata* and also provides paradigms that notably facilitate reusability. However, expressing concurrency in object-oriented languages raises some difficulties. For instance, integrating concurrency together with inheritance may lead to violate the encapsulation property of object-oriented programming. In this paper, we introduce a new strongly-typed, concurrent object-oriented language, called Arche, that has been developed to simplify the construction of reusable distributed applications intended to execute on the object-based system Gothic [Banatre et al.91]. Among other features, the Arche language integrates both inheritance and concurrency in a satisfying way.

1.1 Concurrent Object-Oriented Programming

As notably discussed in [Andrews et al.83], there exist three issues that underlie the design of any concurrent programming model. First, expression of concurrent execution should be defined: for instance, this may be achieved through process declaration or a parallel imperative. Second, it should be chosen if processes will communicate by shared variables or by message passing. Finally, a mechanism allowing expression of synchronization between processes has to be determined. Various design concerns have led to different concurrent object-oriented programming models with respect to these three issues. Let us first briefly examine solutions that have been retained in existing concurrent object-oriented languages for expressing concurrent execution and inter-process communications.

In the Actor model [Agha86], the behavior of an object is understood as a function of its incoming messages. Actors are self-contained, interactive, and independent components that communicate by asynchronous message passing. Another concurrent object-oriented programming model [Agha90] tends to unify a declarative view of objects as abstract data types with a procedural view of objects as sequential processes. In this model, each object is a sequential process responding to messages sent to that object. Languages that use such a model include languages of the Pool family [America89] in which processes communicate by Ada-like rendezvous [Ada83]. A variant of the above model consists in

not systematically associating a process to each object. Objects that contain a process are said to be *active* while other objects are passive data structures. This is the approach taken in the Emerald language [Raj et al.91] in which a thread of control originating in one object may span other objects. The Dragoon language [Atkinson et al.91] takes the same approach for the integration of concurrency. However, this language is not solely based on the object paradigm. Due to the attempt to provide a practical vehicle for encouraging reuse and distribution in a large-scale software system, the designers have enriched the Ada language [Ada83] with the principal features of object-oriented programming. Finally, another approach to concurrent object-oriented programming is to provide the process notion apart from the object notion. Among other solutions, this may be achieved through concurrent programming constructs. Languages that follow this approach include Beta[1] [Kristensen et al.87] and Guide [Krakowiak et al.90]. In the Beta language, objects may communicate by means of synchronized execution of peculiar objects (i.e., objects that are coroutines). In the Guide language, objects communicate via shared objects. An alternative solution to the introduction of the process notion apart from the object notion consists in offering libraries of predefined classes that allow expressing concurrency. This is the approach taken in Modula-3 [Cardelli et al.88] and Presto [Bershad et al.88].

From the perspective of synchronization, most of the above languages integrate mechanisms that have been defined independently of the object paradigm. For instance, the Emerald language that is based on a communication model via shared variables uses monitors [Hoare74]. On the other hand, the Pool language whose communication model is by message passing, offers a synchronization mechanism based on the Ada language one. It appears that the definition of a synchronization mechanism for a concurrent object-oriented language is a difficult problem. Most synchronization mechanisms interfere with inheritance in that local changes in the class hierarchy require the redefinition of synchronization constraints elsewhere in the hierarchy. Proper integration of both inheritance and concurrency control has been a very active area of research and some solutions have been proposed (e.g., [Kafura et al.89, Tomlinson et al.89]). However, to our knowledge, most of them elude the issue of code specialization (i.e., subtyping), which seems to have only been studied in [Nierstrasz et al.91].

[1] Let us notice here that the Beta language is actually based on a unique general abstraction mechanism, called *pattern*.

1.2 Our Approach

The concurrency model of the Arche language has been defined so as to keep all the benefits of the object paradigm. In particular, conditional synchronization is supported without interfering with code reuse nor code specialization. Furthermore, we have adopted an approach similar to the one undertaken in the design of the Hermes language [Strom et al.91]. In the same way, Arche is a simple language for expressing computation and program composition. The compiler together with the underlying object-based system are assigned the task to choose the proper implementation with respect to architecture and performance. Hence, programmers can concentrate on functionality and modularity issues.

The design of Arche follows from the experience that we gained with the strongly-typed, concurrent object-based language Polygoth [Lecler89, Benveniste90, Banatre et al.91]. Polygoth was firstly designed to enable explicit management of distributed and replicated data. An effort was also made to provide convenient notations to deal with distribution. Polygoth is primarily based on the notion of *multiprocedure* [Banatre et al.86], which is the outcome of a generalizing approach to the integration of parallelism and procedures undertaken in [Banatre80]. Multiprocedures extend properties of procedures to the parallel framework. Like procedures, multiprocedures abstract implementation details of (parallel) computations and allow declaration and composition of these computations. A *fragmentation* mechanism is further offered in order to map the multiprocedure notion onto classes. The state of a fragmented object is a collection of data logically distributed, and can only be accessed through method calls. Methods are straightforwardly implemented by multiprocedures. Indeed, multiprocedures enable expression of distributed computations that perfectly match the data fragmentation. Despite the benefits of the undertaken approach for managing distribution, some drawbacks have been presented in [Benveniste et al.91]. In particular, it appears to be non-suited for integrating inheritance. This has led to the definition of the Arche language.

As a successor of the Polygoth language, Arche is aimed at the development of distributed applications. Its design has mainly been guided by our concern with minimizing sources of programming errors, should it be at the expense of other qualitative features such as flexibility and efficiency. Among other consequences, this has led us to define a language belonging to the Algol family with respect to block structure, scope rules and type-checking. The usefulness of types in programming languages is well known. In particular, they allow support-

ing error-detection by either static or run-time type-checking, and, when static analysis can be performed, types enable improving execution efficiency by taking advantage of type information during code generation. Our design choices meant achieving a proper integration of the various mechanisms offered by a concurrent object-oriented language instead of strengthening the expressive power of each mechanism. In particular, provisions for synchronization have been defined so as to keep all the advantages of subtyping and inheritance. In what follows, we do not describe all the features of the Arche language whose detailed definition may be found in [Benveniste et al.92]. We mainly focus on the Arche parallel features and show how a satisfying integration of concurrency with object-oriented programming is established. Moreover, comparisons with related work are provided throughout the paper.

The remainder of this paper is organized as follows. Section 2 first gives an overview of the Arche language: Object declaration, subtyping, and inheritance are briefly described and its basic concurrency model is discussed. Section 3 then introduces the Arche conditional synchronization mechanism, which allows delaying an operation execution until the embedding object is in a proper state. This mechanism is defined in such a way that synchronization constraints may be inherited and specialized. Section 4 describes another synchronization mechanism that enables a larger scale synchronization, that is, more than just a pair of processes may be involved. Such a synchronization has been recognized as worthwhile (e.g., through the barrier notion [Jordan et al.89]) for various frameworks of concurrent programming but has scarcely been examined for object-oriented programming. Finally, assessment of our proposal and future work are discussed in Section 5.

2 Overview of the Arche Language

The Arche language is hybrid; in addition to declaration of object descriptions, usual types (e.g., integer, ordinal, record) are offered. Nevertheless, usual types being almost those defined in the Modula-2 language [Wirth82], they are not discussed here. In the following, we focus on the object model of Arche.

2.1 Types, Classes and Objects

An object declaration is composed of a *view type*, which defines the object interface, and a *class*, which defines the object implementation.

Types are declared by means of constructors. The type constructor `VIEW` defines a view type, which is an entity akin to an abstract data type. View types constitute the means to integrate the notion of class interface within the Arche type system. Among other consequences, this results in definition of highly protected objects since objects can only be accessed through the operations of their interface. The Arche language offers a collection of built-in view types that have a peculiar status. In particular, we identify the *pseudo* view type `SEQUENCE`, used for declaration of sequences (substitute for arrays in Arche), whose methods are usual operations on sequence (e.g., `CAT`, `APPEND`, `FRONT` and `TAIL`). Let us notice here that unlike other view types, sequences do not require definition of classes, objects of type `SEQUENCE` are declared through the use of the constructor `SEQ OF` followed by the type of their elements. This peculiar status of the view type `SEQUENCE` comes from our decision to avoid generic view types in the current Arche definition, this aspect being under study.

Implementation details of view types are declared by means of classes. As a first source of polymorphism, a view type may be implemented by many classes. Thus, two objects of the same type do not necessarily have the same concrete representation. A class declaration names the view type that it implements, and encloses a list of declarations (i.e., types, constants, variables, and procedures) and a body. Procedures declared within a class define method implementations. However, a class may declare more procedures than methods, extra procedures are then qualified as *local* to the class. Given a class declaration, the compiler verifies that the class does implement the specified view type. This boils down to check that for each method declared within the view type, there exists an *associated* procedure with same name and signature within the class. A class cannot contain two procedures with the same name. The class body states actions that have to be carried out when an instance of the class is created through the expression `NEW ClassName (Parameters)`[2]. The class body mainly aims at establishing a proper initial state for the created object. However, should this block be omitted, the instance creation consists only in the initial update of the object variables. Created objects are all persistent, their destruction relies on a garbage collector offered by the underlying object-based system [Puaut92]. An example of view type together with its implementation are provided hereafter

[2]Provisions are also made to allow creation of objects through copy of existing ones. In the rest of this paper, we omit this additional feature whose detailed definition may be found in [Benveniste et al.92].

Example 1

Usual expressions and commands of Arche are similar to the ones defined in Modula-2 [Wirth82]. The following view type *Point* defines a point interface in a two-dimensional coordinate system:

```
Point = VIEW(x, y:  FLOAT)
              GetPos :  () (x, y:  FLOAT);
              Move   :  (dx, dy:  FLOAT) ();
              Rotate :  (alpha:FLOAT) ()
        END;
```

The view type `Point` is parameterized by the value of the point initial position, and provides three methods:

- `GetPos` that takes no parameter and returns the point current position;
- `Move` that takes a translation vector as input parameter and moves the point; and
- `Rotate` that takes the rotation angle as input parameter and processes the requested rotation.

A possible implementation of `Point`[3], considering a cartesian representation, is:

```
CLASS CartesianPoint IMPLEMENTS Point =
   VAR X: FLOAT := x; Y: FLOAT := y;
   PROCEDURE
      GetPos = BEGIN RETURN(X, Y) END GetPos;
      Move = BEGIN X := X + dx; Y := Y + dy END Move;
      Rotate =
         VAR c, s:  FLOAT;
         BEGIN
            c := COS(alpha); s := SIN(alpha);
            X := X*c - Y*s; Y := X*s + Y*c
         END Rotate
   END CartesianPoint;
```

The separation of object declarations into views and classes is not new. In particular, similitudes are to be identified with definition and implementation modules of Modula-2. Such a decomposition offers several advantages. In Arche, a view may be implemented by several classes thus providing a first source of polymorphism. Dually, enabling association of multiple views to one class has been recognized as worthwhile to help software development (e.g., [Shilling et al.89]). Therefore, the notions of view and class provide a sound basis towards the definition of additional mechanisms to help large software management.

[3]In the Arche definition, types that are non-local to a class are actually declared in compilation units called *libraries*. Thus, the name of a non-locally declared type should be preceded by the name of its enclosing library. We have not introduced this aspect in our paper for brevity.

2.2 Subtyping and Inheritance

Subtyping and inheritance although being distinct notions [Cook et al.90] are closely related in Arche; inheritance is only permitted when a heir class implements a subtype of the type implemented by its ancestor.

The subtyping relation, noted <:, follows from the corresponding relation of the programming language Modula-3 [Cardelli et al.89]. In the remainder of this paragraph, we detail only subtyping of view types even though subtyping rules are defined for all the Arche types [Benveniste et al.92]. The subtyping relationship applies on two view types if one is explicitly defined as being an extension of the other. This extension is expressed through prefixing. However, this definition does not stand for the predefined view types `ANY` and `NULL`; `ANY` is the greatest view type while `NULL` is the smallest. Let T_i be a view type, we define:

(S_1) `NULL` <: T_i

(S_2) T_i `VIEW ... END` <: T_i

(S_3) T_i <: `ANY`

Furthermore, two identical types are subtypes of each other:

(S_4) let `T` and `U` be two types, if `T` $\equiv$ `U` then `T` <: `U` and `U` <: `T`

The Arche language defines a *coercion* relation, noted $\hookrightarrow$, in order to allow polymorphic references. This definition directly follows from the one proposed in [Cardelli et al.89] for the Modula-3 language. As an example of coercion rule, we have: *Given an expression* e *of type* $\mathcal{U}$ *and a variable* v *of type* $\mathcal{V}$, *if* $\mathcal{U} <: \mathcal{V}$ *then* e $\hookrightarrow$ v.

Only single inheritance is provided. We have not introduced multiple inheritance, even though useful, due to its interaction with the Arche concurrency model. The inheritance of a class `C` is expressed in the class header by means of the clause `INHERITS C`. However, as previously stated, a class is allowed to inherit of a class `C` only if its type is a subtype of the one implemented by `C`. A subclass may freely access any of the entities declared in its superclass. Procedure redefinition is explicitly specified in the clause **redefines** of the redefining class. Finally, a redefined procedure can be called from its redefining entity through the use of the command **super**.

The Arche inheritance mechanism is very similar to those of most strongly-typed object-oriented languages. Inheritance is restricted to satisfy subtyping,

which results in a type-safe language. An inheritance mechanism that supports more of the flexibility of Smalltalk inheritance [Goldberg et al.83] while allowing static type-checking, has been proposed in [Cook89]. However, this proposal is made in the framework of sequential programming. As an alternative to the inheritance mechanism for code reuse, the compositional model has been proposed in [Raj et al.89] to meet peculiar requirements of the Emerald language. The compositional model is less powerful than the inheritance mechanism. In particular, specialization of objects recursively defined does not seem to be addressed in the compositional model.

2.3 Basic Concurrency Model

From the perspective of concurrency, parallelism is explicit and hence integrates the *process* notion. As discussed earlier, there exist several ways for integrating the process notion in a language based on the object paradigm. Our concern with software correctness has led us to chose a solution similar to the one retained for the languages of the Pool family [America89].

A process is created whenever an object is. Such a process acts on its state as specified by the object class and its external behavior is the one specified by the object type. A process (or object) creation leads to concurrently execute the class body with the creating object. Once the execution of the block terminates, the process waits for a method call.

The communication model is based on method calls. From the programmer's standpoint, a method call corresponds to a traditional procedure call; in particular, the caller is blocked until the *callee* returns. Furthermore, a method call frees the programmer of localizing remote objects. Caller and callee are treated asymmetrically in this communication mechanism. A method call, expressed as `dest ! Method (parameter)`, may appear anywhere an expression or a command is expected depending upon whether the callee returns a result or not. Thus, the programmer chooses where a sending communication occurs in the program, and the receiver is required to be explicitly stated. On the other hand, reception of a method call is implicit and cannot be specified by the programmer; it is controlled by the language semantics. The programmer has no means to select a peculiar call nor can he/she identify the method caller (unless specified as a parameter).

Furthermore, a *n-readers, one-writer* policy is implemented within any object through the notions of *observer* and *modifiers*. Two kinds of procedures may

be distinguished within an object depending upon whether they modify or read the object state: *modifiers* are procedures of the first kind and *observers* are procedures of the second. Distinguishing observers and modifiers requires to be always able to label the nature of nested calls. When calls refer only to procedures local to the enclosing class, this can easily be achieved by the compiler. On the other hand, external calls (i.e., method calls) make complex and expensive such an implicit labeling. Our approach enforces the use of interfaces and hence tends to mask implementation details of view types from their users. Thus, in order to exploit their semantics, observers have to be explicitly declared within view types. This is expressed through the use of the clause `OBSERVER`. The compiler then verifies that procedures implementing observers do not modify the object state. To our knowledge, only languages of the Pool family distinguish these two kinds of operations through the notions of *method* and *routine* [America89]. Such a separation allows a higher degree of concurrency without compromising the association of pre- and post-assertions to procedures as, for instance, in Eiffel [Meyer88], though these assertions are not provided in Arche.

3 Conditional Synchronization

Synchronization mechanisms are needed to control the order in which messages are processed so as to preserve integrity of objects. In particular, although many operations may be defined on an object, some operations may have to be delayed until the object is in a proper state. For instance, considering a bounded buffer, the operation performed to add an element to the buffer can be applied only if and when the buffer is not full. This aspect is referred to as conditional synchronization. As earlier stated, it appears that mechanisms for conditional synchronization often interfere with inheritance. Former studies on the definition of a conditional synchronization mechanism for a concurrent object-oriented language have shown that centralizing synchronization constraints within a procedure prevents these constraints from being inherited by subclasses [Kafura et al.89]. On the other hand, expressing synchronization constraints in a decentralized way seems to be a key approach for integrating both inheritance and concurrency in an object-oriented language. We have therefore retained this solution. More precisely, conditional synchronization is addressed, as in [Kafura et al.89] and [Tomlinson et al.89], by the specification of a mutable set of available methods. Such sets may then be inherited and enriched in subclasses.

For most of them, former work on the definition of a conditional synchro-

nization mechanism for a concurrent object-oriented language consider only typeless languages. Strong typing raises additional difficulties that are related to subtyping. Subtyping notably relies on the principle of substitutability [Wegner et al.88], that is, it enables safe use of an object of type T in the replacement of an object of type U if T is a subtype of U. In the parallel framework, this boils down to use a process in the replacement of another one. Synchronization statements largely determine process behaviors. Therefore, the subtyping relation should be enriched so that process behavior may be taken into account when resorting to the substitutability principle. This has led us to integrate synchronization information within types.

3.1 From Type-States to Synchronization-States

A view type defines the set of methods supported by its objects. Expressing conditional synchronization requires additional means to prevent execution of an object method under a given condition. In other words, even though the type of a variable referring to an object "o" indicates the existence of a method M, one should be able to express delayed processing of incoming calls to M due to "o"'s current state. Restricting execution of "o"'s methods according to the methods previously executed by "o" may be compared to the identification of operations whose application on a variable "v" are semantically correct depending on the operations antecedently performed on "v". As an illustration, let us examine the following program fragments where "a" and "b" are integer variables, and "p" is a pointer:

```
(1)  a   := b + 1;        b   := 5;
(2)  b   := 5;            a   := b + 1;
(3)  p^  := b + 1;        p   := ADR(a);
(4)  p   := ADR(a);       p^  := b + 1;
```

These fragments are both type correct. However, statements of the left column are semantically incorrect. Considering the assignment on line (1), the variable "b" has no assigned value and hence the result of the addition is undefined. In the same way, "p" has no address assigned when used in line (3).

The above example highlights the fact that even though the application of an operation can be syntactically correct, it may sometimes result in semantically incorrect programs. In order to detect and reject such programs at compile-time, the notion of *type-state* has been introduced in [Strom et al.86] for strongly-typed languages. In this proposal, a type is not only characterized by the set of

operations it provides but also by an automaton. The automaton associated to a type T is defined as follows: states indicate the subset of T's operations that may actually be applied, and transitions correspond to applications of T's operations. This additional characterization of types results in a finer static checking that allows detection of semantic errors.

Our concern here is to characterize states under which a given method call may be selected. We enrich the definition of view types by sets of peculiar type-states, called *synchronization-states*. Within a view type T, a synchronization-state "s" defines the methods of T that may be executed by objects of type T when they are to select an incoming message, according to the sequence of methods that they have previously performed. Syntactically, synchronization-states of a view type are declared in the clause STATE written as:

$$\texttt{STATE } \mathtt{s}_1\texttt{: } \{\mathtt{M}_{1_1}, \ldots, \mathtt{M}_{1_{k_1}}\}; \ldots; \mathtt{s}_n\texttt{: } \{\mathtt{M}_{n_1}, \ldots, \mathtt{M}_{n_{k_n}}\}.$$

The above declaration associates methods $\mathtt{M}_{i_j}$, $1 \leq j \leq k_i$, to the synchronization-state $\mathtt{s}_i$, $1 \leq i \leq n$. After its creation, an object is, by default, in the synchronization-state that includes all the methods of its type. However, a set of initial synchronization-states may be specified in the view header. The execution of a method may modify the object synchronization-state only if and when the method terminates. We then say that a *state transition* occurred and the synchronization-state reached by this transition is called a method's *post-state*. A method may initiate various state transitions, the set of method's post-states is declared in the embedding view type by means of the clause POST, written as:

$$\texttt{POST } \mathtt{M}_1\texttt{: } \{\mathtt{s}_{1_1}, \ldots, \mathtt{s}_{1_{k_1}}\}; \ldots; \mathtt{M}_m\texttt{: } \{\mathtt{s}_{m_1}, \ldots, \mathtt{s}_{m_{k_m}}\}.$$

The above clause means that the execution of the method $\mathtt{M}_i$ (assuming that $\mathtt{M}_i$ terminates), $1 \leq i \leq m$, leads the enclosing object to one of the synchronization-states $\mathtt{s}_{i_j}$, $1 \leq j \leq k_i$, for the next method call. The clause POST is a requisite whenever a clause STATE is declared. However, not all the view methods are required to appear within the clause POST. The absence of a method M means that M does not change the object's synchronization-state. Finally, when a view type T does not declare a clause STATE, it means that any method call can always be selected and hence that no state transition takes place while the object executes. In other words, this implies that objects of type T are free of conditional synchronization. We now illustrate the notion of synchronization-state.

Example 2

We define the view type **Semaphore** as:

```
Semaphore = VIEW (i:  CARDINAL) {open, close}
                P     : () ();
                V     : () ();
             STATE
                open  : {P, V};
                close : {V};
             POST
                P     : {open, close};
                V     : {open}
          END;
```

The clause **STATE** declares two synchronization-states: **open** that allows execution of methods **P** and **V**, and **close** that allows only executing **V**, any caller of **P** being blocked until the enclosing object reaches the synchronization-state **open**. The clause **POST** indicates that the execution of **P** leads the enclosing object "o" to the synchronization-state **open** or **close** depending on the value of "i" and on the operation previously executed by "o". On the other hand, executing **V** always leads the enclosing object to the synchronization-state **open**. Finally, the initial synchronization-state of a semaphore object may be either **open** or **close** depending on the value passed as parameter for "i".

Care must be taken to preserve the consistent substitution of processes. The subtyping rules given in Subsection 2.2 for view types have to be enriched in order to support specialization of synchronization-states and post-states.

3.2 Subtyping and Synchronization-States

Let us illustrate declaration of a subtype of a view carrying synchronization information through an adaptation of the example discussed in [Kafura et al.89].

Example 3

We define a view type **Buffer** that provides the interface of a bounded buffer whose elements are of type **T**.

```
Buffer = VIEW (n:  INTEGER)  empty
              Put      :  (i:  T) ();
              Get      :  () (i:  T);
           STATE
              full     :  {Get};
              empty    :  {Put};
              partial  :  {Get, Put};
           POST
              Put      :  {partial, full};
              Get      :  {partial, empty}
        END;
```

We now want to define a view type, called **BufQueue**, that allows removing the buffer last element. This is easily achieved by extending **Buffer** by the method *GetRear*:

```
BufQueue = Buffer VIEW ()
              GetRear:  () (i:  T);
           POST
              GetRear:  {partial, empty};
           END;
```

Unless modified, this solution is not satisfactory because the method **GetRear** cannot be executed; none of the synchronization-states includes this method.

With respect to the above example, it appears that the subtyping rule S_2 (see § 2.2, page 9) has to be enriched in order to allow establishing synchronization constraints of a subtype and reporting them to the supertype. Such constraints have to be settled according to the substitutability principle. Ideally, a refinement relationship would have to be defined. Such a relation would allow definition of a correspondence between the synchronization properties of a subtype and the less precise ones of its supertype. However, as we have not yet foreseen how refinement of synchronization constraints could be achieved, and compelled by a type-checker, the current Arche definition offers a weaker enforcement of the substitutability principle. Given an instance "o" of a class implementing a subtype of T, the definition of subtyping ensures that synchronization of "o" may always exhibit a behavior (i.e., the trace of methods it executes) common with the one of objects of type T. Considering the extension of a view type T, this criteria is met by the three following rules.

(R_1) An existing state of T may be redefined but this redefinition may only be achieved through extension;

(R_2) A new state may be defined but it has to be defined as an extension of a state already defined in T;

(R_3) Post-states of T's methods may be extended.

Syntactically, extension of synchronization-states (resp. post-states) is expressed in the corresponding view clause by declaring methods (resp. states) to be added to the existing synchronization-state (resp. method's post-states). As shown below, rule R_1 enables modification of example 3 so that the method GetRear can be executed, while rules R_2 and R_3 cope with definition of additional synchronization-states.

Example 4

Using rule R_1, type **BufQueue** can be completed as follows:

```
BufQueue = Buffer VIEW ()
                GetRear :  () (i:  T);
             STATE
                full    :  {GetRear};
                partial :  {GetRear};
             POST
                GetRear :  {full, partial}
          END
```

States **full** and **partial** are extended by the method **GetRear**. The initial post-state of **BufQueue** is the one of its supertype, no post-state appears in the view header. Should initial post-states be mentioned in a subtype view, the initial post-state of the view type would then be the union of these with those of the supertype.

Let us now consider definition of a subtype **ReverseBuffer** of **Buffer** that offers method **Swap**. This method exchanges the two first elements of the buffer and thus must only execute when the buffer size is greater than one. Type **ReverseBuffer** has then to define a new synchronization-state.

```
ReverseBuffer = Buffer VIEW
                    Swap       :  () ();
                 STATE
                    atLeastTwo :  partial:  {swap};
                 POST
                    Swap       :  {atLeastTwo};
                    Put        :  {atLeastTwo};
                    Get        :  {atLeastTwo}
              END;
```

State **atLeastTwo** extends state **partial** by method **Swap** (rule R_2); definition of **POST** specifies that this state is the post-state of **Swap** and furthermore belongs to post-states of **Put** and **Get** (rule R_3).

Given the definition of the conditional synchronization mechanism introduced so far, expression and inheritance of synchronization constraints within classes directly follow.

3.3 Conditional Synchronization Within Classes

Within view type implementations (i.e., classes), state transitions are expressed through the use of the command BECOME followed by the name of the synchronization-state to be reached. A state transition expressed within an operation (e.g., a procedure) becomes effective only if and when the operation terminates. The compiler is assigned the task to check whether synchronization-states that may be reached during a method (resp. object initialization) execution are only those defined in the clause POST of the view type for the corresponding method (resp. in the view header). There is no restriction on the invocation of BECOME within an operation. Should more than one state transition occur when an operation executes, only the last transition is meaningful. On the other hand, if no transition takes place, the object's synchronization-state remains. Expression of state transitions is illustrated hereafter, we provide an implementation of the view type **ReverseBuffer** (see example 4).

Example 5

We first propose an implementation of **Buffer**:

```
CLASS CBuffer IMPLEMENTS Buffer =
   CONST max = n;
   VAR in, out:  INTEGER; tamp:  SEQ OF T;
   PROCEDURE
      Put = BEGIN
            tamp[in] := i; in := (in + 1) MOD max;
            IF in = (out + 1) MOD max
               THEN BECOME full ELSE BECOME partial
            END
         END Put;
```

```
      Get = BEGIN
            i := tamp[out]; out := (out + 1) MOD max;
            IF in = out
               THEN BECOME empty
               ELSE BECOME partial
            END;
            RETURN(i)
         END Get;
   BEGIN
      in := 0; out := 0; BECOME empty
END CBuffer;
```

A possible implementation of **ReverseBuffer** is then:

```
CLASS CReverse IMPLEMENTS ReverseBuffer =
   INHERITS cBuffer;
   REDEFINES Put, Get;
   PROCEDURE
      Put = BEGIN
            SUPER(i);
            IF ABS(in-out) > 1 THEN BECOME atLeastTwo END
         END put;
      Get = BEGIN
            i := SUPER();
            IF ABS(in-out) > 1 THEN BECOME atLeastTwo END;
            RETURN(i)
         END Get;
      Swap =
         VAR x:  T;
         BEGIN
            x := tamp[in];
            tamp[in] := tamp[(in+1) MOD max]; tamp[(in+1) MOD max] := x
         END Swap;
   BEGIN
      SUPER()
END CReverse;
```

The conditional synchronization mechanism that we have proposed has been designed so as to keep all the benefits of subtyping and inheritance. Most of the existing proposals do not address the impact of synchronization on subtyping. Furthermore, existing object-oriented languages often prevent inheritance of synchronization constraints, and require to rewrite synchronization code. An overview of these proposals follows.

3.4 Related Work

To our knowledge, stating synchronization information within types has been scarcely examined. A notion of type for active objects has been proposed in [Nierstrasz et al.91]. In the same way, the proposed notion is defined to meet requirements of substitutability in the presence of active objects. However, even though some resemblance may be identified, it should be noted that our notion of view type has been elaborated independently of the above research. Furthermore, this work does not seem to have been integrated in a programming language.

The interference of synchronization with inheritance has formerly been investigated in [Kafura et al.89]. This reference has guided us in the definition of our synchronization mechanism, notably for the use of the command **BECOME**. However, let us point out that the mentioned reference does not address issues related to subtyping, this proposal being made in the framework of an Actor-based language. Still in the framework of Actor-based languages, a general strategy has been proposed in [Agha86] for solving synchronization problems. It consists in explicitly buffering messages that are not ready to be processed. Once the Actor is ready to handle the message, it resends the message to itself. This approach is not fully satisfactory in that *inherited* methods have to be modified to deal with additional ones. Another approach to synchronization in Actor-based languages has been proposed in [Tomlinson et al.89] through the notion of *enabled-set*. This proposal offers similitude with the synchronization-state notion. In the same way, enabled-sets define messages that may be executed in the next state of the object, and they can be inherited and composed. Two properties are further introduced on enabled-sets: synchronization specifications can be passed in messages and parameterized on the basis of the message content. However, unlike our approach, the issue of subtyping is ignored due to the target language, only inheritance is addressed.

Let us now examine synchronization mechanisms offered by strongly-typed, concurrent object-oriented languages. In the Pool family languages [America89], each class defines a body that implements the synchronization constraints for all methods defined in the class. The code dispatches or delays the execution of a message depending on the state of the object. Pool does not support inheritance of class body and thus of synchronization constraints [America et al.89]. Besides, commands stating acceptance of incoming messages are all over the code, and a new acceptance command would have to be written in the inherited code for each new method added. In the Guide language, synchronization is expressed by means

of a control clause enclosing *activation conditions* [Decouchant et al.89]. An activation condition determines the states of an object that permit the execution of a message for a given method. Such conditions are based on synchronization counters [Robert et al.77], which allow boolean expressions dependent upon the number of messages that have been received, completed, or in execution. An activation condition may also refer to any state variables of the receiver and to the message content. Inheritance of synchronization constraints appears to be restrictive in Guide; a control clause is either inherited or not. Thus, inheritance may lead to rewrite a part of the control clause already declared in the superclass. The Hybrid language [Nierstrasz87] provides a construct, called *delay-queue*, that may be opened or closed to permit messages to be executed or not. When an operation is defined, it may be associated with a delay-queue. As messages arrive, they are placed in the appropriate delay-queue based on the operation that the message is requesting. There is at least one shortcoming in the hybrid approach, closing or opening new methods within inherited methods require modification of their code.

4 Multi-Party Synchronization

Synchronization involving more than two participants has been recognized as a useful tool for concurrent programming. The multiway rendezvous (e.g., [Francez89]) and the barrier notion [Jordan et al.89] are examples of mechanisms for multi-party synchronization. Other proposals introduce multi-party synchronization mechanisms that further define abstract parallel computations. For instance, the Script mechanism hides low-level details that implement patterns of communication [Francez et al.86], and the multiprocedure notion generalizes the procedure notion to the parallel framework [Banatre et al.86] (see § 1.2). However, to our knowledge, existing multi-party synchronization mechanisms have been designed in the context of a fixed network. Therefore, they do not deal with dynamic concurrency where processes are dynamically generated, destroyed, or reconfigured. Furthermore, the issue of multi-party synchronization does not seem to have been addressed in the framework of strongly-typed, concurrent object-oriented programming despite its benefits for processing object states in a concurrent framework.

The simplest structure of an object state is given by a list of variables where some variables may be typed by a view. Such a declaration enables definition of various graphs whose nodes have a fixed arity. However, in practice, nodes with

non-fixed arity are often needed. Even though such nodes may be characterized by means of a linked list, a direct representation would facilitate access to peer nodes. The predefined view type SEQUENCE allows such a representation. For instance, we may declare the type of a figure as: `Figure = SEQ OF Polygon` where `Polygon` is defined as follows.

```
Polygon = VIEW ...
              Move:  (v:  POS) ();
              ...
          OBSERVER
              Draw:  () ();
              ...
      END;
```

Let us now consider the movement of a figure. We want to express that all the figure components move in an interdependent way. As there is no need for an ordered treatment, this may be achieved in parallel. Given the notations introduced so far, concurrent management of sequence elements may be achieved through the creation of processes so as to implement asynchronous communications. This solution is not satisfactory. Its expression leads to awkward code, which is even worst in the presence of sharing. As a sequence is not handled as a whole but at the element level, modifying a shared sequence requires additional synchronization. For instance, let us consider that a figure is owned by two users that both want to move it. It is then necessary to explicitly serialize operations applied on the figure's elements[4]. In the same way, additional synchronization is needed when series of operations are to be performed on a sequence. For instance, let us consider a programmer that wants to move a figure and then draw it on a screen. Explicit serialization is required if one does not want to draw the figure until *all* its elements have been moved.

4.1 From Multiprocedures to Multi-Operations

In order to allow simple management of sequences, we define a new kind of call that allows invocation of methods at the sequence level. A sequence of objects embeds the predefined sequence operations and the methods of its composing objects. Sequence methods that are inherited from its composing objects are to be related to *multiprocedures* [Banatre et al.86, Banatre et al.89]. Thus, considering

[4]Let us remark here that the same problem would apply if the figure was moved sequentially.

a sequence "s" of type SEQ OF T, "s"'s methods that are inherited from T are called *multi-operations*[5], and define a peculiar kind of multiprocedures.

A multi-operation signature is defined so as to express parameter passing to each of its components; the signature is obtained by applying the type constructor SEQ OF to each parameter type of the corresponding method declared in the base type. For instance, given a variable F of type Figure, the signature of the multi-operation Move of F is: Move: (SEQ OF POS) (). Let now "p" be a variable of type SEQ OF POS, a call to the multi-operation Move of F is written as F ! Move(p) and is carried out as follows:

- All the objects belonging to F are synchronized;
- Input parameters are distributed among F's components so that the i^{th} element of each actual parameter is given to the multi-operation's i^{th} component;
- Move is executed in parallel by all the components of F;
- Objects are synchronized and their respective contributions to multi-operation result -if any- are collected;
- Results -if any- are built out of each component contribution and made available to the caller;
- Objects of F become available to execute further calls.

Finally, notice that if F is shared, the sequence cannot be modified until the call completes. As F is an object, calls to F referring either to a multi-operation or a predefined sequence operation are serialized. Still considering our example, the movement of a figure F composed of four polygons, followed by its display is expressed as:

```
PROCEDURE MoveFigure: (v: POS) () =
   BEGIN
      F ! Move(<v, v, v, v>); F ! Draw
   END MoveFigure;
```

This solution has still some drawbacks when considering parameter passing to the multi-operation Move; the number of translation vectors has to be equal to

[5]We have not retained the term *multi-method* in order to avoid confusion with the notion of *multi-method* existing in *multiple-dispatching* languages.

the sequence cardinality. This significantly decreases the potential reusability of code; should the number of figure components be modified, the multi-operation call has to be rewritten, which further requires re-compilation of the enclosing class. We introduce a syntactic sugar that allows expressing multicast of a parameter. Within a multi-operation call, when an expression of type T is passed as a parameter while an expression of type SEQ OF T is expected, the expression value is *multicasted* to all the multi-operation components. The above example thus becomes:

```
PROCEDURE MoveFigure:  (v:  POS) () =
   BEGIN
      F ! Move(v); F ! Draw
   END MoveFigure;
```

The proposed implementation of MoveFigure is still not satisfactory if some elements of F are shared. Calls to these elements can be interleaved with the execution of multi-operations Move and Draw. The *coordinated call* [Banatre et al.86] offered by the multiprocedure notion enables preventing such a behavior.

4.2 The Notion of Coordinated Call

A *coordinated call* that may be issued by any multi-operation is a natural extension of the method call mechanism. All the multi-operation components join together to call a multi-operation, and are all synchronized. When the call is terminated, results -if any- are made available to all caller's components before their parallel activities resume. Naturally, the coordinated call for a single element sequence is the traditional method call. The relationship between a calling sequence and the called sequence is a *many in many* nesting. A coordinated call is expressed by stating COCALL before the call expression. Furthermore, pseudo-variables are introduced so that components may specify information with respect to their role in the embedding multi-operation (or sequence).

When a method executes as a component of a multi-operation, it may refer to the corresponding sequence through the use of the pseudo-variable WE. For instance, when an object of F executes Move in response to F ! Move(v), WE refers to F. However, should an object "o" execute a method apart from a multi-operation, accessing WE refers to the sequence <o>. Finally, the order of a given object within a sequence may be known by using ME hence leading to the equality WE(ME) = SELF within any procedure. We are now able to modify the implementation of MoveFigure. The following method MoveDraw:

```
MoveDraw: (v: POS) () =
   BEGIN Move(v); COCALL WE ! Draw END MoveDraw;
```

is added within Polygon and MoveFigure is now written as:

```
PROCEDURE MoveFigure: (v: POS) () =
   BEGIN F ! MoveDraw(v) END MoveFigure;
```

Let us remark here that the above modification proposed for Polygon is simplified due to the inheritance mechanism. This example furthermore suggests specialization of view types to use objects as sequence elements.

4.3 Related Work

The multi-operation facility may be related to linguistic constructs proposed in the literature for concurrent programming. The *FT-Actions* [Jalote et al.86] allows composition of parallel computations. However, the composition can only be achieved through *static* nesting and the number of calling components should be equal to the one of the callee. The *Script* notion [Francez et al.86] also requires the number of calling components to be equal to the one of the callee; nonetheless, the nesting of Scripts is *dynamic*. Compared to the above constructs, the multi-operation notion enables expression of a more general nesting (i.e., n components within m). Furthermore, the multi-operation notion is defined to cope with dynamic concurrency while existing proposals, including the multiprocedure notion, are made in the context of a fixed network. The strong synchronization offered by the coordinated call may be related to the *barrier* notion [Jordan et al.89]. In the same way, notations are introduced to express that processes have to wait for each other prior to continue their execution. However, the barrier notation is a low level synchronization primitive while the multi-operation notion allows definition of abstract parallel computations. Finally, Actor-based languages allow expressing concurrent evaluation of the arguments of a functional form [Agha86]. A join continuation is further provided to synchronize the evaluation of the different arguments. Despite its functional nature, this last form of synchronization is very close to the barrier notation.

5 Conclusions

In this paper, we have presented a new strongly-typed, concurrent object-oriented language, called Arche, intended to support the development of distributed applications.

5.1 Assessment of our Proposal

In the Arche language, mutual exclusion is implicit; methods that modify the object state (i.e., modifiers) are serialized while others (i.e., observers) may be executed concurrently. Compared to existing proposals, this may seem as a restrictive choice. However, this facilitates formal setting of object properties since association of pre- and post-assertions to procedures according to the object state becomes possible.

Provisions for conditional synchronization have been defined so as to keep all the advantages of subtyping and inheritance. In particular, information related to conditional synchronization is stated within the object type. More precisely, we have introduced a new mechanism for conditional synchronization, which is based on the type-state notion [Strom et al.86]. Availability of object methods at a given time is defined by synchronization-states that are declared in the object type. A synchronization-state "**s**" specifies the set of methods that may be executed when an object is in state "**s**". This additional characterization of types relates subtyping to the behavior of resulting processes. Thus, when an object is to be used in place of another one through the use of subtyping, its behavior as a process is also determinant in the validity of the substitution. Independently of our research, integration of synchronization information within types for active objects has been studied in [Nierstrasz et al.91]. However, to our knowledge, this work has not been incorporated within a programming language. As for conditional synchronization mechanisms, the most related to ours are those of [Kafura et al.89] and [Tomlinson et al.89]. Nonetheless, these proposals are made in the framework of the Actor model and thus avoid the issue of subtyping, only inheritance of synchronization properties is addressed. Finally, as discussed previously, most strongly-typed, concurrent object-oriented languages do not support straight inheritance of synchronization properties. Moreover, none of these proposals considers the impact of concurrency control on subtyping. Integration of synchronization information within view types has led us to restrict inheritance to satisfy subtyping; a class can inherit from another class `C` only if it implements a subtype of the type implemented by `C`.

The Arche language supports application of operations on a collection of objects through the multi-operation notion, which is based on the multiprocedure notion [Banatre et al.86]. This facility permits declaration and composition of parallel computations hence resulting in a general dynamic nesting of n components within m. To our knowledge, such a facility has never been addressed

in the framework of strongly-typed, concurrent object-oriented programming despite its ability to simply manage collection of peer objects. Furthermore, in addition to the benefits of the multi-operation notion discussed in this paper and deeper examined in [Benveniste92], it also offers advantages from the perspective of fault-tolerance. Study of error recovery in asynchronous systems has led to identify the decomposition of a parallel application into parallel sub-actions as essential for the design of fault-tolerant distributed software [Campbell et al.86]. As multi-operations can be composed through coordinated calls, a large parallel operation may be defined in terms of smaller ones. Furthermore, replication has been recognized as worthwhile to enforce fault-tolerance in a distributed system. Multi-operations allow simple management of copy consistency. Among other solutions, a straightforward implementation consists in invoking a multi-operation any time a replicated entity is modified, the invoked sequence being composed of all the objects that own a copy. Benefits of the multi-operation notion for fault-tolerance are further increased due to the Arche exception handling mechanism [Issarny92] that provides basic support for writing robust and reusable concurrent applications. We have not detailed this aspect in our paper for brevity. The interested reader is referred to the above reference in which the use of the Arche exception handling mechanism and multi-operation notion to program robust, distributed applications is examined.

5.2 Future Work

A compiler for the Arche language has been implemented in the framework of the INRIA/Bull project Gothic at the research institute IRISA (Rennes, France). The compiler generates C code that is intended to execute on the object-based system Gothic [Banatre et al.92]. The Gothic system notably provides complex built-in operations to help management of Arche parallel features (e.g., multi-operations, coordinated call). The current implementation of Arche has still to be enhanced to support efficient and fair execution of multi-operations. This is a difficult problem that requires further research prior to be solved. In particular, we have to design an efficient protocol to solve concurrent multi-operation calls whose respective destination sequences share common objects. This problem may be seen as a generalization of the problem that underlies the effective implementation of the input-output construct of CSP [Buckley et al.83].

Throughout this paper, we have emphasized our concern with software correctness. Formal semantics along with good specification, verification and validation

techniques for Arche appears to be an attractive area for future research. In particular, the issue of software correctness has to be investigated. We have pointed out that the n-readers/1-writer policy implemented within objects provides a sound basis towards the definition of assertions for procedures. However, the Arche language does not enforce this feature unlike, for instance, the Eiffel language [Meyer88]. Moreover, composition of object properties so as to define properties of a whole application has also to be examined. Definition of a proof theory for a concurrent object-oriented language (i.e., a subset of Pool [America89]) has been proposed in [America et al.88]. Nonetheless, this work eludes some language features; for instance, the original Pool communication model is replaced by CSP-like communications. Besides examining formal aspects, we are also concerned with integration of additional mechanisms within Arche so as to provide the programmer with better supports for development of large software.

ACKNOWLEDGMENTS: The authors would like to thank J.P. Banâtre, M. Banâtre, M. Jegado, P. Lecler. I. Puaut, J.P. Routeau and H. Ruiz Barradas for helpfull discussions on the subject of this paper. V. Issarny acknowledges the department of computer science and engineering of the University of Washington for receiving her as a visiting scholar.

References

[Ada83] Ada. – *The Programming Language ADA.* – Springer Verlag, 1983, *Lecture Notes in Computer Science*, volume 155.

[Agha86] Agha (G.). – *Actors: A Model of Concurrent Computation in Distributed Systems.* – MIT press, 1986.

[Agha90] Agha (G.). – Concurrent object-oriented programming. *Communications of the ACM*, vol. 33 (9), 1990, pp. 125–141.

[America et al.88] America (P.) and de Boer (F.). – *A Proof System for a Parallel Language with Dynamic Process Creation.* – Technical report, Eindhoven, The Netherland, Philips Research Laboratory, 1988.

[America et al.89] America (P.) and Van Der Linden (F.). – *A Parallel Object-Oriented Language with Inheritance and Subtyping.* – Tech-

nical report, Eindhoven, The Netherland, Philips Research Laboratory, 1989.

[America89] America (P.). – Issues in the design of parallel object-oriented language. *Formal Aspects of Computing*, vol. 1 (4), 1989, pp. 366–411.

[Andrews et al.83] Andrews (G. R.) and Schneider (F. B.). – Concepts and notation for concurrent programming. *ACM Computing Surveys*, vol. 15 (1), 1983, pp. 3–43.

[Atkinson et al.91] Atkinson (C.), Goldsack (S.), Di Maio (A.) and Bayan (R.). – Object-oriented concurrency and distribution in Dragoon. *Journal of Object-Oriented Programming*, vol. 4 (1), 1991, pp. 11–19.

[Banatre et al.86] Banâtre (J. P.), Banâtre (M.) and Ployette (F.). – The concept of multi-functions, a general structuring tool for distributed operating system. *In : Proceedings of the Sixth Distributed Computing Systems Conference.*

[Banatre et al.89] Banâtre (J. P.) and Benveniste (M.). – Multiprocedures: Generalized procedures for concurrent programming. *In : Proceedings of the Third Workshop on Large Grain Parallelism.*

[Banatre et al.91] Banâtre (J. P.) and Banâtre (M.), editors. – *Les systèmes distribués : l'expérience du système Gothic.* – InterEditions, 1991.

[Banatre et al.92] Banâtre (M.), Belhamissi (Y.), Bryce (C.), Puaut (I.) and Routeau (J. P.). – *Gothic: A Distributed Object-Based System.* – Research Report, Rennes, France, IRISA, 1992. In preparation.

[Banatre80] Banâtre (J. P.). – *Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables.* – Rennes, France, Thèse d'état, Université de Rennes I, 1980.

[Benveniste et al.91] Benveniste (M.) and Issarny (V.). - Le modèle de programmation de Gothic. *In : Les systèmes distribués : l'expérience du système Gothic*, pp. 55–79. - InterEditions, 1991.

[Benveniste et al.92] Benveniste (M.) and Issarny (V.). - *Arche : un langage parallèle à objets fortement typé.* - Research Report 1646, Rennes, France, INRIA, 1992.

[Benveniste90] Benveniste (M.). - *Operational Semantics of a Distributed Object-Oriented Language and its* Z *Formal Specification.* - Research Report 1230, INRIA, 1990.

[Benveniste92] Benveniste (M.). - *Procédure, parallélisme et objet : une approche par la généralisation.* - Rennes, France, Thèse de doctorat, Université de Rennes I, 1992. In preparation.

[Bershad et al.88] Bershad (B. N.), Lazowska (E. D.) and Levy (H. M.). - Presto: A system for object-oriented parallel programming. *Software - Practice and Experience*, vol. 18 (8), 1988.

[Buckley et al.83] Buckley (G. N.) and Silberschatz (A.). - An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, vol. 5 (2), 1983, pp. 223–235.

[Campbell et al.86] Campbell (R. H.) and Randell (B.). - Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, vol. SE-12 (8), 1986, pp. 811–826.

[Cardelli et al.88] Cardelli (L.), Donahue (J.), Glassman (L.), Jordan (M.), Kalsow (B.) and Nelson (G.). - *Modula-3 Report.* - Technical report, Palo Alto, California, USA, Digital Systems Research Center, 1988.

[Cardelli et al.89] Cardelli (L.), Donahue (J.), Jordan (M.), Kalsow (B.) and Nelson (G.). - The Modula-3 type system. *In : Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 202–212.

[Cook et al.90] Cook (W. R.), Hill (W. L.) and Canning (P. S.). – Inheritance is not subtyping. *In : Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 125–135.

[Cook89] Cook (W.). – A proposal for making Eiffel type-safe. *The Computer Journal*, vol. 32 (4), 1989, pp. 305–311.

[Decouchant et al.89] Decouchant (D.), Krakowiak (S.), Meysembourg (M.), Riveil (M.) and de Pinat (X. Rousset). – A synchronization mechanism for typed objects in a distributed system. *SIGPLAN Notices*, vol. 24 (4), 1989, pp. 105–107. – Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming.

[Francez et al.86] Francez (N.), Hailpern (B.) and Taubenfeld (G.). – Script: A communication abstraction mechanism and its verification. *Science of Computer Programming*, vol. 6, 1986, pp. 35–88.

[Francez89] Francez (N.). – Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, vol. 32 (5), 1989, pp. 235–242. – Corrigenda in Information Processing Letters vol. 35, pp. 57.

[Goldberg et al.83] Goldberg (A.) and Robson (D.). – *Smalltalk-80: The Language and its Implementation.* – Addison-Wesley series in Computer Science, 1983.

[Hoare74] Hoare (C. A. R.). – Monitors: An operating system structuring concept. *Communications of the ACM*, vol. 17 (10), 1974, pp. 549–557.

[Issarny92] Issarny (V.). – *An Exception Handling Mechanism for Parallel Object-Oriented Programming: Towards the Design of Reusable, and Robust Distributed Software.* – Research Report 673, Rennes, France, IRISA, 1992. To appear in Journal of Object-Oriented Programming.

[Jalote et al.86] Jalote (P.) and Campbell (R. H.). – Atomic actions for fault-tolerance using CSP. *IEEE Transactions on Software Engineering*, vol. SE-12 (1), 1986, pp. 59–68.

[Jordan et al.89] Jordan (H.), Benten (M.), Alaghband (G.) and Jakob (R.). – The Force: A highly portable parallel programming language. *In : Proceedings of the International Conference on Parallel Processing*, pp. II–112–II–117.

[Kafura et al.89] Kafura (D. G.) and Lee (K. H.). – Inheritance in actor based concurrent object-oriented languages. *The Computer Journal*, vol. 32 (4), 1989, pp. 297–303.

[Krakowiak et al.90] Krakowiak (S.), Meysembourg (M.), Nguyen van (H.), Riveil (M.) and Roisin (C.). – Design and implementation of an object-oriented strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, vol. 3, 1990, pp. 11–22.

[Kristensen et al.87] Kristensen (B. B.), Madsen (O. L.), Moller-Pedersen (B.) and Nygaard (K.). – The Beta programming language. *In : Research Directions in Object-oriented Programming.* – MIT Press, 1987.

[Lecler89] Lecler (P.). – *Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en œuvre dans le système Gothic.* – Thèse de doctorat, Rennes, France, Université de Rennes I, 1989.

[Meyer88] Meyer (B.). – *Object-Oriented Software Construction.* – Prentice-Hall International, 1988.

[Nierstrasz et al.91] Nierstrasz (O.) and Papathomas (M.). – Towards a type theory for active objects. *ACM OOPS Messenger*, vol. 2 (2), 1991, pp. 89–93.

[Nierstrasz87] Nierstrasz (O.). – Active objects in Hybrid. *In : Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications.*

[Puaut92] Puaut (I.). – Distributed garbage collection of active objects with no global synchronization. *In : Proceedings of the International Workshop on Memory Management.* pp. 148–164. – Springer Verlag.

[Raj et al.89] Raj (R. K.) and Levy (H. M.). – A compositional model for software reuse. *The Computer Journal*, vol. 32 (4), 1989, pp. 312–322.

[Raj et al.91] Raj (R. K.), Tempero (E.), Levy (H. M.), Black (A. P.), Hutchinson (N. C.) and Jul (E.). – Emerald: A general purpose programming language. *Software Practice and Experience*, vol. 21 (1), 1991, pp. 91–118.

[Robert et al.77] Robert (P.) and Verjus (J.P.). – Toward autonomous descriptions of synchronization modules. *In : Proceedings of the IFIP Congress.* – North Holland.

[Shilling et al.89] Shilling (J. J.) and Sweeney (P. F.). – Three steps to views: Extending the object-oriented paradigm. *In : Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 353–361.

[Strom et al.86] Strom (R. E.) and Yemini (S.). – Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, vol. SE-12 (1), 1986, pp. 157–171.

[Strom et al.91] Strom (R. E.), Bacon (D. F.), Goldberg (A. P.), Lowry (A.), Yellin (D. M.) and Yemini (S. A.). – *Hermes: A Language for Distributed Computing.* – Prentice-Hall, 1991.

[Tomlinson et al.89] Tomlinson (C.) and Singh (V.). – Inheritance and synchronization with enabled-sets. *In : Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 103–112.

[Wegner et al.88] Wegner (P.) and Zdonik (S. B.). – Inheritance as an incremental modification mechanism or what like is and isn't like. *In : Proceedings of the European Conference on*

Object-oriented Programming. pp. 55–77. – Springer Verlag. Lecture Notes in Computer Science, volume 322.

[Wirth82] Wirth (N.). – *Programming in Modula-2.* – Springer Verlag, 1982.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 683 TARGET TRACKING BY VISUAL SERVOING
Aristide S. SANTOS, François CHAUMETTE
Octobre 1992, 50 pages.

PI 684 UNE DESCRIPTION LINEAIRE COMPLETE ET IRREDONDANTE DU POLYTOPE ASSOCIE AU PROBLEME DU VOYAGEUR DE COMMERCE ASYMETRIQUE A 6 SOMMETS
Reinhardt EULER, Hervé LE VERGE
Octobre 1992, 30 pages.

PI 685 MISE EN CORRESPONDANCE DE SEGMENTS DANS UNE SEQUENCE D'IMAGES PAR UNE APPROCHE LOCALE
Samia BOUKIR, Patrick BOUTHEMY, François CHAUMETTE, Didier JUVIN
Octobre 1992, 30 pages.

PI 686 FROM EQUATIONS TO HARDWARE. TOWARDS THE SYSTEMATIC MAPPING OF ALGORITHMS ONTO PARALLEL ARCHITECTURES
François CHAROT, Patrice FRISON, Eric GAUTRIN, Dominique LAVENIER, Patrice QUINTON, Charles WAGNER
Octobre 1992, 18 pages.

PI 687 THE COMPILATION OF PROLOG and its Execution with MALI
Pascal BRISSET, Olivier RIDOUX
Novembre 1992, 90 pages.

PI 688 GENERALISATION DE L'ANALYSE FACTORIELLE MULTIPLE A L'ETUDE DES TABLEAUX DE FREQUENCE ET COMPARAISON AVEC L'ANALYSE CANONIQUE DES CORRESPONDANCES
Lila ABDESSEMED, Brigitte ESCOFIER
Novembre 1992, 34 pages.

PI 689 OVERVIEW OF THE KOAN PROGRAMMING ENVIRONMENT FOR THE iPSC/2 AND PERFORMANCE EVALUATION OF THE BECAUSE TEST PROGRAM 2.5.1..
François BODIN, Thierry PRIOL
Décembre 1992, 16 pages.

PI 690 CONCURRENT PROGRAMMING NOTATIONS IN THE OBJECT-ORIENTED LANGUAGE ARCHE
Marc BENVENISTE, Valérie ISSARNY
Décembre 1992, 34 pages.

PI 691 COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORIES
Masaaki MIZUNO, Michel RAYNAL, Gurdip SINGH, Mitchell L. NEILSEN
Décembre 1992, 24 pages.

ISSN 0249-6399